Science of Computer Programming 75 (2010) 311–332

Contents lists available at ScienceDirect

Science of Computer Programming

journal homepage: www.elsevier.com/locate/scico

Decision-making coordination and efficient reasoning techniques for feature-based configuration

Marcilio Mendonca *, Donald Cowan

David R. Cheriton School of Computer Science, University of Waterloo, Canada

ARTICLE INFO

Article history:
Received 23 April 2008
Received in revised form 5 August 2009
Accepted 9 September 2009
Available online 5 December 2009

Keywords:
Software Product Lines
Product configuration
Feature models
Feature modelling
Decision-making coordination
Automated reasoning
Constraint-based reasoning

ABSTRACT

Software Product Lines is a contemporary approach to software development that exploits the similarities and differences within a family of systems in a particular domain of interest in order to provide a common infrastructure for deriving members of this family in a timely fashion, with high-quality standards, and at lower costs.

In Software Product Lines, feature-based product configuration is the process of selecting the desired features for a given software product from a repository of features called a *feature model*. This process is usually carried out collaboratively by people with distinct skills and interests called *stakeholders*. Collaboration benefits stakeholders by allowing them to directly intervene in the configuration process. However, collaboration also raises an important side effect, i.e., the need of stakeholders to cope with *decision conflicts*. Conflicts arise when decisions that are locally consistent cannot be applied globally because they violate one or more constraints in the feature model.

Unfortunately, current product configuration systems are typically single-user-based in the sense that they do not provide means to coordinate concurrent decision-making on the feature model. As a consequence, configuration is carried out by a single person that is in charge of representing the interests of all stakeholders and managing decision conflicts on their own. This results in an error-prone and time-consuming process that requires past decisions to be revisited continuously either to correct misinterpreted stakeholder requirements or to handle decision conflicts. Yet another challenging issue related to configuration problems is the typically high computational cost of configuration algorithms. In fact, these algorithms frequently fall into the category of NP-hard and thus can become intractable in practice.

In this paper, our goal is two-fold. First, we revisit our work on *Collaborative Product Configuration* (CPC) in which we proposed an approach to describe and validate collaborative configuration scenarios. We discuss how collaborative configuration can be described in terms of a workflow-like plan that safely guides stakeholders during the configuration process. Second, we propose a preliminary set of reasoning algorithms tailored to the feature modelling domain that can be used to provide automated support for product configuration. In addition, we compare empirically the performance of the proposed algorithms to that of a general-purpose solution. We hope that the insights provided in this paper will encourage other researchers to develop new algorithms in the near future.

* Corresponding address: David R. Cheriton School of Computer Science, University of Waterloo, 200 University Ave. W., N2L 3G1 Waterloo, ON, Canada.
E-mail addresses: marcilio@csg.uwaterloo.ca (M. Mendonca), dcowan@csg.uwaterloo.ca (D. Cowan).

0167-6423/$ – see front matter © 2009 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2009.12.004

1. Introduction

Software Product Lines (SPLs) [1,2] is a contemporary approach to software development that exploits the similarities and differences within a family of systems in a particular domain of interest in order to provide a common infrastructure for deriving members of this family (a.k.a. software products) in a timely fashion, with high-quality standards, and at lower costs. Research in the SPL field has been very intense in the last decade and has been translated to important advances in the software industry quite successfully.

In SPLs, products are derived from a common set of core assets (e.g., source code, UML models, test cases, documentation, software libraries) to attend to the needs of a particular customer or market segment. The term "feature" has been commonly used as an abstraction for the core assets, and form the building blocks for describing products in the product line. That is, each product is represented by a unique combination of features. Specifically, a feature is "*a prominent and distinctive aspect or characteristic that is visible to various stakeholders*" [3]. However, because not all kinds of feature combinations are legal (e.g., features may be mutually-exclusive), *feature models* [3,4] have been proposed as a means to describe feature constraints that prevent illegal combinations from being derived. Therefore, only those combinations that do not violate the constraints in the feature model are considered legal. Currently, several SPL approaches [3,5,4,6] and tools [7–9] support the notion of feature models.

Feature-based product configuration (from now on simply called *product configuration*) is the process of selecting the desired features for a given software product. This process is usually carried out collaboratively by people with distinct skills and interests: the *stakeholders*. Collaboration benefits stakeholders by allowing them to directly intervene in the configuration process. However, collaboration also raises an important side effect, i.e., the need of stakeholders to cope with *decision conflicts* properly. A conflict occurs when two or more local decisions (e.g., selection of feature *A*, deselection of feature *B*) cannot be applied in a global context because they violate one or more constraints in the feature model. In this case, one or more decisions need to be revisited and eventually changed. Automated support to assist stakeholders with handling decision conflicts is highly desirable.

Unfortunately, current product configuration systems in SPLs are typically single-user-based in the sense that they do not provide means to coordinate concurrent decision-making on the feature model. As a consequence, configuration is carried out by a single person that is in charge of representing the interests of all stakeholders and managing decision conflicts on their own. This results in an error-prone and time-consuming process that requires past decisions to be revisited continuously either to correct misinterpreted stakeholder requirements or to manage decision conflicts. We claim that single-user-based configuration is inadequate as it puts a heavy burden on a single individual and does not provide a systematic approach for anticipating and handling decision conflicts. Moreover, single-user-based configuration approaches do not scale and are barely adequate for large scale scenarios such as those observed in the automotive domain in which product lines encompass tens of thousands of features [10].

Yet another challenging issue related to configuration problems is the high computational cost of configuration algorithms [11]. These algorithms frequently fall into the category of NP-hard and thus can become intractable in practice. For instance, the problem of automatically computing a solution for a decision conflict is typically translated to the problem of finding a solution to a *satisfiability problem* which is well-known to be NP-complete [12]. As a consequence, researchers have been forced to adapt existing general algorithms to a specific problem domain in order to make them tractable within that domain. We claim that the same strategy has to be applied in the SPL domain, specifically to address product configuration problems where efficient algorithms are needed to boost automated configuration tools.

In this paper, our goal is two-fold. First, we revisit our work on *Collaborative Product Configuration* (CPC) [13] in which we proposed an approach to describe and validate collaborative configuration scenarios. We discuss how collaborative configuration can be described in terms of a workflow-like plan that safely guides stakeholders during the configuration process. We provide an illustrative example of the approach. Second, we propose a preliminary set of reasoning algorithms tailored to the feature modelling domain that can be used to provide automated support for product configuration. In addition, we compare empirically the performance of the proposed algorithms to that of a general-purpose solution. Our work provides valuable insights on the development of efficient reasoning algorithms for the feature modelling domain that will hopefully encourage other researchers to build new algorithms in the near future.

This paper is organized as follows. Section 2 provides background on feature models and their relation to constraint satisfaction problems. Our approach to collaborative product configuration is presented in Section 3. Section 4 discusses the infrastructure needed to support automated reasoning about configuration problems. A discussion is carried out to examine how domain knowledge can be used to boost the performance of general-purpose constraint solvers in the context of configuration. In addition, results of experiments are shown to compare the efficiency of the proposed algorithms with those of a general-purpose solutions. Prototype tools to support the CPC approach and the reasoning techniques are discussed in Section 5. Section 6 covers related work and Section 7 concludes the paper.

2. Background

2.1. Feature models

A feature model allows product line designers and domain analysts to describe features and specify constraints to enforce legal feature combinations. Feature models were initially proposed by a domain analysis approach called Feature-Oriented

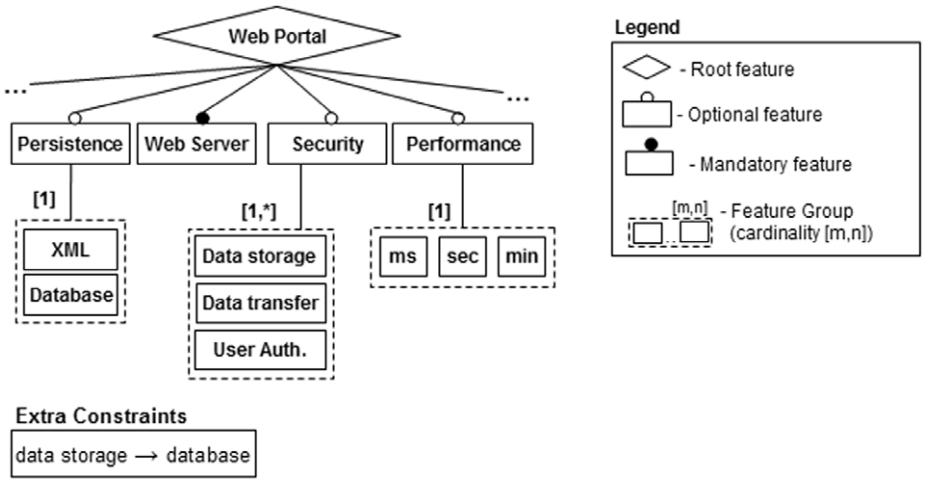

Fig. 1. Partial feature model for a Web portal product line.

Domain Analysis (FODA) [3]. Since then, feature models have been extended in several ways, mostly to adapt the original proposal to specific research needs.

Fig. 1 shows a partial feature model for a Web portal product line. Two structures can be distinguished in the model: the *feature tree* (top left-hand side) and the *extra constraints* (bottom left-hand side). The feature tree represents a hierarchical arrangement of the product line features using a pre-defined set of feature types which causes the feature relations to be much easier for humans to understand. Five types of features are possible: mandatory, optional, inclusive-OR, exclusive-OR, and the root feature. The root feature is usually called the *concept*, as it models the variabilities associated with a particular domain concept. In the figure, the root feature is represented by a diamond-shaped element labelled Web Portal. Rectangles represent features. Rectangles decorated with an unfilled circle on top represent optional features (e.g., Persistence, Performance). Optional features can only be selected if their parent feature is selected. Rectangles decorated with a filled circle represent mandatory features (e.g., Web Server). A mandatory feature is always selected with the selection of its parent feature. Feature groups are represented by dashed rectangles enclosing two or more (grouped) features. Cardinalities with lower and upper bounds are attached to feature groups to indicate mutual-exclusion relations (e.g., [1], [1,*]). In this paper, we consider feature models with only two kinds of feature groups: inclusive-OR (cardinality [1,*]) and exclusive-OR (cardinality [1]) groups. In an inclusive-OR group, at least one grouped feature must be selected if the parent feature is selected (e.g., features Data Storage and Data Transfer can be selected simultaneously if feature Security is selected). Meanwhile, only one grouped feature of an exclusive-OR group can be selected if the parent feature is selected (e.g., feature Ms is selected and features Sec and Min are deselected if parent feature Performance is selected). If a parent feature is deselected, all its descendants must be deselected.

The extra constraints represent additional relations attached to the feature tree. For instance, consider extra constraint (Data Storage → Database) (we read *Data Storage requires Database*) in Fig. 1. It specifies that a database system must be available in a product whenever data storage security is a requirement. We define *extra constraint representativeness* (or ECR) of a feature model as the ratio of the number of (non-repeated) features in the extra constraints and the number of features in the feature tree. For instance, the ECR for a feature model containing 100 features, 12 of which are used in the extra constraints, is 0.12 or 12%(12 ÷ 100).

A product can be specified by selecting features in the feature model. For instance, a legal product specification for the partial Web portal feature model in Fig. 1 is S_1 = (Web Portal, Persistence, XML, Web Server). However, specification S_2 = (Web Portal, Persistence, XML, Database, Web Server) is *illegal* since features XML and Database are mutually-exclusive but appear together in the specification. Notice that, by convention, the root feature is part of any legal product specification.

During product configuration, the goal is to derive a legal specification by selecting and deselecting features in the feature model and without violating the constraints in the model. In CPC, the goal is also to achieve a valid configuration but in a collaborative process. More details are provided in a later section.

2.2. Feature models and constraint satisfaction

A *Boolean Constraint Satisfaction Problem* (B-CSP) is a triple $\langle X, D, C\rangle$, where X is a set of variables over domain $D = \{0, 1\}$, and C is a set of constraints on those variables. Every constraint $c_i \in C$ restricts the combined values of its variables, denoted by $V(c_i)$. An assignment $A(S)$ is a set of tuples $\langle s_i, v_i\rangle$ such that $S \subseteq X$, $s_i \in S$, $v_i = 0$ *or* 1, and s_i appears at most once in $A(S)$. We say that $A(S)$ satisfies a constraint $c_i \in C$, if $V(c_i) \subseteq S$ and the assignments made to c_i's variables in $A(S)$ cause this constraint to evaluate to 1 (*true*). A solution to the B-CSP is an assignment $A(X)$ that satisfies all constraints in C. The problem of finding a solution for a B-CSP is commonly referred to as *satisfiability* or SAT. SAT is well-known to be an NP-complete problem [12].

A *Boolean formula* (or propositional formula) can be used to encode a B-CSP. Such a formula is constructed using Boolean variables and the operators $\vee$ (or), $\wedge$ (and), $\rightarrow$ (implication), $\leftrightarrow$ (bi-implication), and $\neg$ (not). For example, $a \rightarrow (b \vee c)$ is a Boolean formula. A formula is said to be in CNF (conjunctive normal form) if it represents a conjunction of clauses, where a clause is a disjunction of literals. A literal is either a variable or its negation. For instance, formula $\phi = (a \vee b) \wedge (a \vee \neg c) \wedge (\neg c \vee \neg d)$ is in CNF and contains variables a, b,c, and d.

A constraint solver (general constraint system) or a SAT solver (Boolean constraint system) can be used to check whether there is a solution for a given SAT problem. Currently, there are several efficient solvers freely available [14,15] supported by state-of-the-art algorithms resulting from decades of research in the artificial intelligence field.

It is known that a feature model can be straightforwardly translated to a Boolean formula representation [16]. In the translation process, features in the feature model become variables in the SAT problem and the relations in the feature tree and in the extra constraints turn into constraints in the problem. For full details on the translation rules please refer to [16].

If the translation rules are applied to the partial feature model in Fig. 1 then formula ψ below is obtained. Formulas Eqs. (1)–(8) represent feature tree relations while formula Eq. (9) corresponds to the single extra constraint. Notice that we use the special operator *XOR* to represent a mutual-exclusion operation on two or more variables such that exactly one of such variables has to be *true*.

$$\psi = (\text{Web Portal}) \wedge \tag{1}$$
$$(\text{Persistence} \rightarrow \text{Web Portal}) \wedge \tag{2}$$
$$(\text{Web Server} \rightarrow \text{Web Portal}) \wedge \tag{3}$$
$$(\text{Security} \rightarrow \text{Web Portal}) \wedge \tag{4}$$
$$(\text{Performance} \rightarrow \text{Web Portal}) \wedge \tag{5}$$
$$(\text{Persistence} \leftrightarrow \mathit{XOR}(\text{XML, Database}) \wedge \tag{6}$$
$$(\text{Security} \leftrightarrow (\text{Data Storage} \vee \text{Data Transfer} \vee \text{User Auth})) \wedge \tag{7}$$
$$(\text{Performance} \leftrightarrow \mathit{XOR}(\text{Ms, Sec, Min})) \wedge \tag{8}$$
$$(\text{Data Storage} \rightarrow \text{Database}). \tag{9}$$

Formula ψ represents a formal encoding of the Web portal feature model that enables the use of off-the-shelf constraint solvers to reason about the model. For instance, if the formula is unsatisfiable we can conclude that the feature model is *inconsistent*. Likewise, a solution to the formula represents a legal configuration in the feature model. A decision conflict in the feature model can be viewed as a variable assignment that does not satisfy formula ψ, and a solution to the conflict as a set of changes to the assignment that make it satisfiable. For this reason, constraint solvers have been viewed as an attractive and rich infrastructure for developing configuration systems. However, as constraint solvers are designed to be general solutions their performance within a given domain might be poor or unacceptable. In Section 4, we look into a domain-specific approach that combines the strengths of constraint solvers with the efficiency of domain-specific algorithms to provide a complete infrastructure for reasoning on configuration problems. Before delving into the technical algorithmic discussion, we first introduce our approach to collaborative product configuration.

3. Collaborative product configuration

Fig. 2 depicts two configuration scenarios. Scenario (A) illustrates a traditional non-collaborative configuration approach in which stakeholders provide requirements to the product manager, who in turn interprets and translates them into configuration decisions. In this context, stakeholders are passive in the configuration process. A feature model serves as input to the configuration process and as a result a complete valid product specification is produced. Automated support is desired to assist the product manager with reasoning about his decisions and with propagating decisions throughout the feature model. Scenario (B) describes our approach to coordinate human decision-making in product configuration. The approach consists of two phases. In phase-1 (scenario B1), the goal is to produce a *configuration plan* that describes the configuration tasks and the order in which they should be carried out. Phase-2 represents the actual configuration process where stakeholders make configuration decisions guided by the plan created in phase-1. In both phases, participants should ideally be supported by automated tools.

The first step in phase-1 is called *splitting*. It aims at partitioning the decisions in the feature model into smaller more-manageable units called *configuration spaces*. Configuration spaces can be configured by different groups of stakeholders. There are rules that must be followed to correctly split the feature model. For instance, the splitting must cover the entire feature model space and the configuration spaces specified cannot overlap any decisions. These rules should be enforced by an automated system supporting the splitting process. The concept of a *decision* is related to that of a feature, specifically, a decision sets the state of a feature to either*selected* (included in the product specification) or *deselected* (excluded from the product specification).

Once the feature model is split into valid configuration spaces, a step known as *plan creation* takes place. At this step, the product manager devises a workflow-like configuration plan based on the splitting. The plan specifies a set of *configuration sessions* and their order of execution. Configuration sessions are assigned to configuration actors (e.g., stakeholders) and

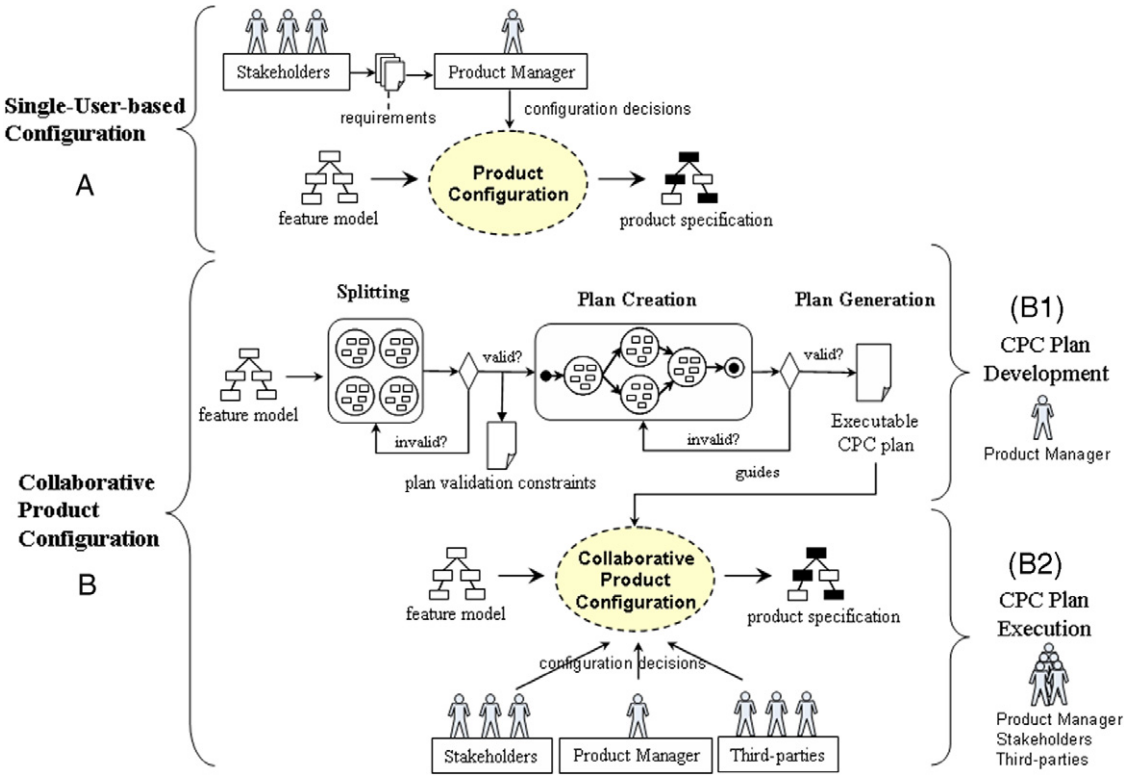

Fig. 2. Single-User-based (A) and collaborative (B) product configuration scenarios.

can contain one or more configuration spaces. The order in which the sessions are arranged in the plan (e.g., sequentially, concurrently) is defined by the product manager but is also subject to validation rules. The validation of the plans is critical as invalid plans can lead to incorrect product specifications. Validating plans involves inspecting dependencies among tasks. We distinguish between strong and weak dependencies. A strong dependency requires that tasks are arranged in a sequence. Meanwhile, weak dependencies allow tasks to run in parallel. Since interdependent configuration sessions can be carried out concurrently, a special kind of session called a *merging session* is defined to handle potential decision conflicts, i.e., conflicts caused by decisions made in distinct yet dependent configuration sessions. A merging session is only necessary if two or more parallel interdependent sessions contain decisions that together violate a relation in the feature model. During the merge, configuration actors in charge of those sessions reason about potential solutions to the conflict, supported by automated tools. The final step in phase-1 is the *plan generation* in which an executable encoding representing the CPC plan is generated, i.e., the high-level plan description is converted into a machine-executable format (e.g., a workflow described in XML).

Once the CPC plan is validated and generated, phase-2 (scenario B2) is initiated. Phase-2 represents the actual product configuration process that aims at producing a valid product specification by configuring the feature model. The difference from scenario B1 is that now multiple configuration actors are allowed to directly participate in the configuration process. The plan created in phase-1 is used to guide this process.

3.1. Plan development phase

This section illustrates phase-1 of the CPC approach using the Web portal product line depicted in Fig. 3. Notice that Fig. 3 shows an expanded version of the feature model in Fig. 1 in which constraint (Data Storage → Database) was removed and several other relations were added as extra constraints.

3.1.1. Splitting responsibilities

The product manager role is responsible for the splitting phase since they have a privileged view of the stakeholders and their expertise. Additionally, the product manager can anticipate potential conflicting situations and try to avoid them. Fig. 3 shows a possible splitting for the Web portal product line. Nine configuration spaces are depicted: Wp, St, Pe, Sc, Pf, Se, Ad, Ws, and Pr. Notice that some features appear in more than one configuration space (e.g. Ad Server, Protocols), contradicting what we said before. In fact, the overlapping is only allowed for features called *junction points*. A junction point is a feature that connects a parent configuration space to one or more child configuration spaces. A single (parent) configuration space contains a junction point feature as a leaf node, while for all other (child) spaces the feature is the root of the tree contained

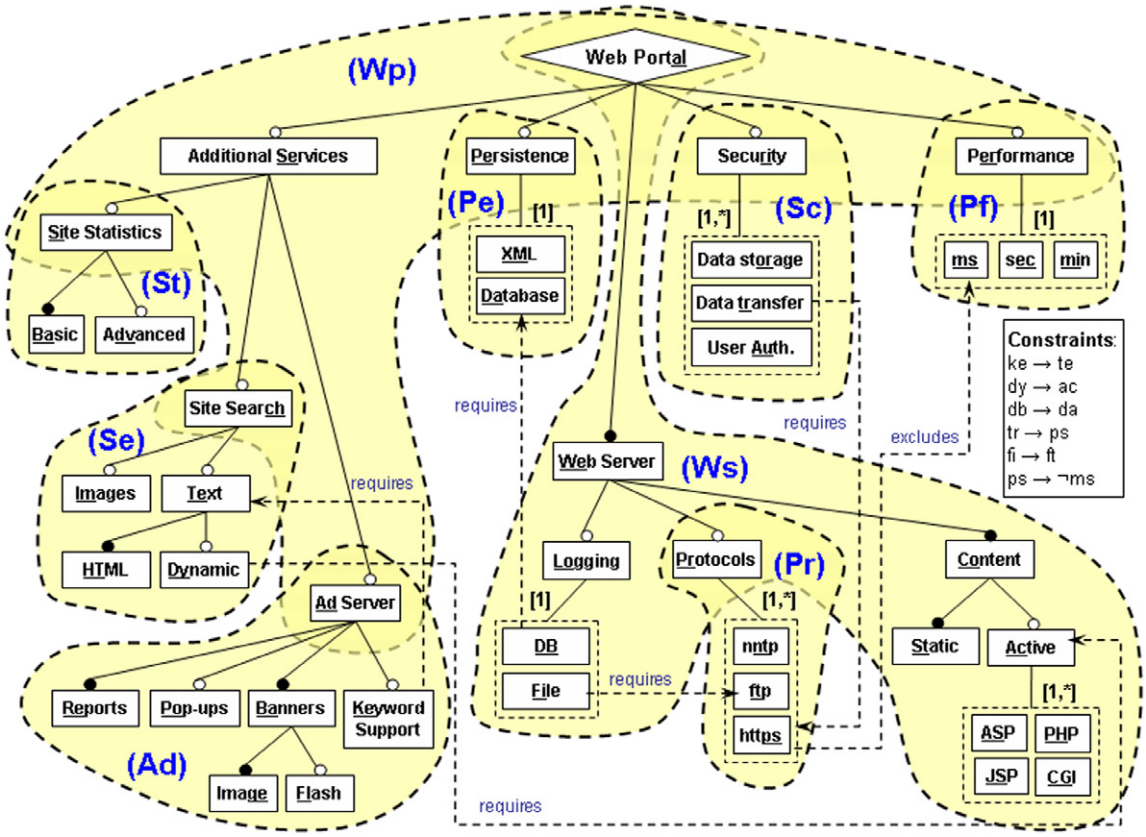

Fig. 3. Feature model for a Web portal product line decorated with configuration spaces (dashed lines).

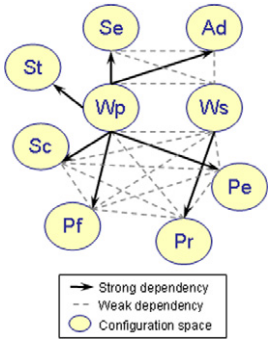

Fig. 4. Merged view of strong and weak dependencies.

in the space. For instance, feature Protocols is part of configuration space Ws and Pr — a leaf node of Ws and the root node of Pr. Hence, Ws is the parent configuration space of Pr. The notion of parent–child spaces is specially important to define the order in which the decisions in these spaces will be addressed. In this sense, configuration spaces can be viewed as clusters of the feature model as their arrangement must respect the hierarchy of the feature tree.

Two kinds of configuration space dependencies are relevant: strong and weak. A configuration space A is *strongly-dependent* on a configuration space B if a single decision in A can potentially impact all decisions in B. From this definition we can conclude that child configuration spaces are always strongly-dependent on their parent spaces. For instance, child space St is strongly-dependent on Wp because if feature Site Statistics is set to *false* all features, and thus decisions, in St are automatically deselected. Two configuration spaces A and B are *weakly-dependent* if some decisions in A can impact some decisions in B, and vice-versa (e.g., Ws and Pe because of constraint DB → Database). Weak dependencies are specified by the extra constraints attached to the feature model.

Fig. 4 shows a merged view of the strong and weak dependencies among the configuration spaces of the Web portal product line. Arrows indicate strong dependencies and weak dependencies are represented by dashed lines. For instance, configuration spaces St, Se, Ad, Pe, Pf and Sc strongly depend on configuration space Wp, since the former are children of

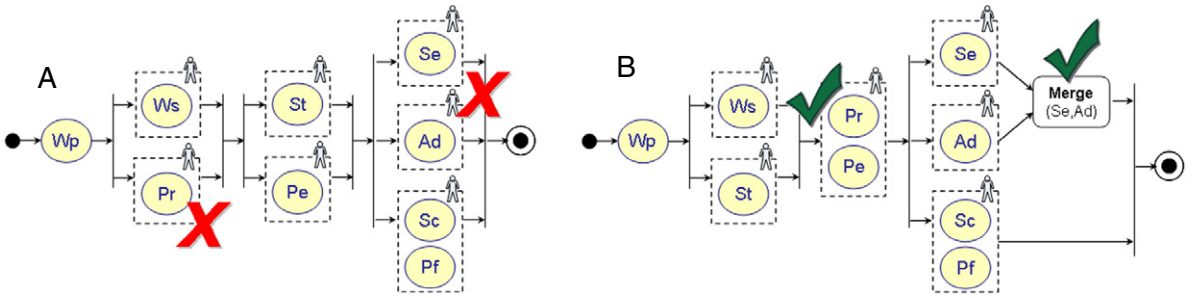

Fig. 5. Invalid (A) and valid (B) CPC plans for the Web portal product line.

the latter. Weak dependencies are identified by performing a dependency analysis of the feature model. In a previous work [17], we discussed a technique to automate the identification of strong and weak dependencies among configuration spaces.

3.1.2. Plan creation

Once the feature model is split into several hierarchical configuration spaces, a plan is specified to group configuration spaces in configuration sessions and to arrange the sessions in sequential and parallel flows. Notice that invalid plans are possible, thus validation rules are required to enforce their correctness. A plan is invalid if it leads to inconsistent product specifications, i.e., specifications that contain mutually-exclusive features. To validate plans we consider the following rules: (1) Whenever a configuration space B is strongly dependent on a configuration space A, A must precede B; (2) If two configuration spaces A and B are weakly dependent they can be arranged either in sequence or in parallel but immediately followed by a merging session. Rules (1) and (2) only apply to configuration spaces placed in different configuration sessions. That is, configuration spaces of the same session are configured by the same team of configuration actors and eventual conflicts are resolved locally.

The configuration plan is a workflow-like structure that groups configuration spaces into configuration sessions and arranges configuration sessions in sequence or parallel. Merging sessions follow dependent configuration sessions, i.e., configuration sessions that contain interdependent configuration spaces. Fig. 5 depicts two configuration plans A and B for the Web portal product line based on the splitting shown in Fig. 3. Plan A is invalid because configuration spaces Ws and Pr are placed in parallel configuration sessions yet Pr is strongly dependent on Ws. Similarly, configuration spaces Se and Ad are also placed in parallel sessions, but because they are weakly dependent on each other a merging session is required to enforce that eventual decision conflicts in those spaces will be addressed.

Plan B fixes the problems of plan A by moving configuration space Pr to a new configuration session that follows Ws configuration session. Similarly, a merging session was added immediately after the configuration sessions of configuration spaces Se and Ad. Finally, note that configuration space St was moved to the same configuration session as configuration space Ws for optimization purposes, since those spaces exhibit no dependencies on each other. The same optimization strategy could have been applied to configuration spaces Ws and Wp.

3.1.3. Plan generation

The last step prior to the actual product configuration process is to generate an executable representation for the CPC plan. Notice that plan B in Fig. 5 is in fact a compact representation of the collaborative configuration process, since many configuration sessions are in fact optional as they depend on decisions made on previous sessions. For instance, even though configuration space St follows configuration space Wp, St configuration session will only be executed if feature Site Statistics is selected during Wp configuration session.

In fact, prior to the execution of any configuration session the underlying workflow system needs to check whether at least one root feature of one configuration space in the session is *true*, otherwise the session is skipped. We say the CPC plan represents a pessimistic view of the collaboration process in which all sessions are executed and decision conflicts arise. In the actual configuration process, many configuration and merge sessions may be skipped as a consequence of previous decisions. We refer to the expanded CPC workflow as the actual executable workflow representation used to augment a CPC plan.

4. Efficient infrastructure for product configuration

Up to this point, we have discussed collaborative configuration in terms of high-level process models for describing and guiding human decision-making. In this section, we switch the focus to the infrastructure required to automate support for CPC. As discussed earlier, several steps during the configuration process require the support of software tools given the complexity of human tasks. For instance, checking whether a particular feature model is consistent, i.e., has at least one valid configuration, requires inspecting all feature model constraints and either finding a contradiction (inconsistency case) or a solution to the model (consistency case). This task is too complex for humans to cope with, especially for large software

product lines. In addition, a particular configuration actor might want to count the number of legal configurations in a partially configured configuration space. This task typically requires searching exhaustively for all solutions of the Boolean formula induced by the feature model, and thus can hardly be performed manually. In fact, counting configurations can be a very time- and space-consuming task even for machines. Yet, in other situations when designing a feature model a domain analyst needs to continuously check the correctness of the developing model. One such check involves detecting and removing "dead" features, i.e., features that can never be part of any product in the product line. This is obviously undesirable since all features in the model should be part of at least one product. Once again, tool assistance is crucial for detecting and eliminating "dead" features.

As we argued before, providing that feature models can be straightforwardly translated to an equivalent Boolean formula, a general constraint solver can be used to assist humans with performing the complex tasks aforementioned. However, as we also argued, constraint solvers are "too-general" solutions, causing those systems to perform poorly for certain domains. The approach pursued by researchers to address this issue has been to either show (empirically or theoretically) that the problems they were dealing with were tractable, or tailor algorithms to the domain of interest. We focus on the latter approach by giving the first step towards the construction of tailored algorithms for the feature modelling domain.

In this section, we propose a domain-specific approach to reason about feature models and product configuration. The approach is based on the use of domain-specific algorithms that explore properties of the feature modelling domain, and the integration of these algorithms with a general constraint-based solution.

4.1. A hybrid solver for feature configuration

What is special about feature models is the fact that they provide two distinct languages for specifying constraints: the feature tree and the extra constraints. The extra constraints represent the conjunction of arbitrary Boolean formulas and therefore it is hard to make any assumption regarding the structure of those formulas. As a result, there is not much that can be done to improve the performance of a general solver regarding the extra constraints. On the other hand, Boolean formulas derived from the feature tree follow an interesting hierarchical arrangement which causes them to hold properties that do not apply in the general case. For instance, any formula derived from a feature tree is necessarily satisfiable (more details later in this section). This is obviously not the case for general formulas. Also, it is possible to count the solutions in the feature tree without having to visit each solution individually but rather by simply applying mathematical operations. Again, this is not the case for general formulas.

In the following section, we explore this fact by proposing algorithms that can be applied exclusively to the Boolean formula induced by the feature tree. The algorithms are aggregated into a domain-specific reasoning system for feature trees. Following, we show how the proposed domain-specific system can be integrated with a general-purpose solver that handles the formula derived from extra constraints, giving rise to a complete "hybrid" solver that is capable of reasoning about the entire feature model formula. Later in this section, we show empirically that the performance of the hybrid solver can be significantly better than that of a pure constraint solver.

4.1.1. FTCS: The feature tree constraint system

We propose a domain-specific constraint solver called *FTCS* (Feature Tree Constraint System) that tailors reasoning algorithms for feature trees. The solver operates on feature trees conforming to the meta-model shown in Fig. 6.

The meta-model uses the *Node* element to indicate that each node in the feature tree must be uniquely identifiable (*id* attribute) and can optionally have a name (*name* attribute). The *RootFeature* element represents the single root feature of the tree. Mandatory and optional features are represented by the *SolitaireFeature* element. The *type* attribute is an enumeration that indicates whether the solitaire feature is optional or mandatory. The *GroupedFeature* element represents features that are part of inclusive-or and exclusive-or groups. The *FeatureGroup* element enforces that a set of grouped features are part of the same group. Attributes *min* and *max* refer to the minimum and maximum cardinality of the group, respectively. We only consider the cases of exclusive-OR (see cardinality [1] in Fig. 1) and inclusive-OR (see cardinality [1,*] in Fig. 1) groups as enforced by the OCL constraint at the bottom of the figure. For $max = -1$ we assume that *max* corresponds to the total number of features in the group. The relation between *GroupedFeature* and *FeatureGroup*, labelled *parent*, enforces that only feature groups can be parent nodes of grouped features. The *ParentNode* and *ChildNode* elements are connected by relation *parent* that indicates that the root feature as well as grouped, mandatory and optional features, all descendants of *ParentNode*, can be parent nodes of *ChildNode* elements such as feature groups, mandatory and optional features. The root node does not have a parent node. The *FeatureTree* element represents a feature tree containing a single root node. The *FeatureModel* element indicates that a feature model always has a single feature tree (*FeatureTree*) and optionally an extra constraint (*ExtraConstraint*). The extra constraint consists of one or more Boolean formulas (*PropositionalFormula*) described textually by attribute *formula*. Notice that the feature model depicted in Fig. 3 conforms to the meta-model in Fig. 6 (annotations such as configuration spaces not considered).

In the following, we examine the operations implemented by the *FTCS* as summarized in Table 1.

Assigning and resetting values. Operation *FT-assign* assigns a truth value to a feature in the feature tree. This operation can be triggered, for instance, by a user performing configuration actions on a feature tree such as selecting (*true* assignment) or deselecting (*false* assignment) features.

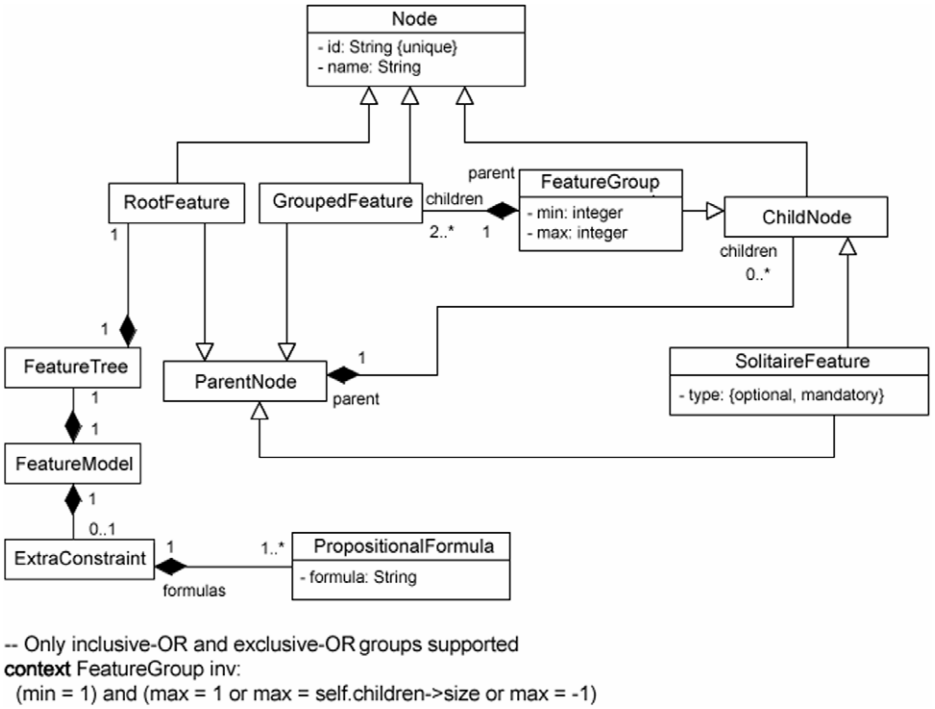

Fig. 6. The meta-model for feature models.

Table 1
FTCS Operations.

Operation	Description
FT-assign(f,v)	Assign the truth value v to feature f; f becomes instantiated
FT-reset(f)	f is unassigned any previously assigned value; f becomes uninstantiated
FT-save-state(s)	Save the current state of the feature tree; state is identified by s
FT-recover-state(s)	Recover the state of the feature tree to state s
FT-propagate(f,v)	Propagate the assignment of v to f throughout the feature tree
FT-is-satisfiable(f)	Returns *true* if the feature tree rooted by f is satisfiable or *false*, otherwise
FT-count-sol(f)	Returns the number of solutions in the feature tree rooted by f

Algorithm 1 Assign feature f the truth value v

Inputs:
f: feature to be assigned a value
v: truth value to be assigned to f

Function **FT-assign**(*f:feature*, *v:Boolean*)
1: **if** (f is uninstantiated) **then**
2: $f \leftarrow v$
3: **else if** (current value of f is different from v) **then**
4: {Error: Assignment Conflict!}
5: **end if**

Algorithm 1 implements the *FT-assign* operation. If the feature is already assigned a different truth value the operation raises an exception to indicate an assignment conflict (lines 3–4) otherwise the value is assigned to the feature (line 2).

FT-reset simply resets the assignment made to a feature so that the feature becomes uninstantiated again. The implementation of *FT-reset* is trivial and thus was omitted.

Saving and recovering system states. The *FT-save-state* (Algorithm 2) and *FT-recover-state* (Algorithm 3) operations save and restore the state of the feature tree, respectively. For each instantiated feature in the feature tree its name and truth value are saved and associated with a unique identifier. The identifier can be used to restore the feature to a particular saved state. These operations are particularly important in the integration of the *FTCS* to a general-purpose constraint solver as will be shown later.

Algorithm 2 Save system state to a state named *s*

Inputs:
s: state name

Function **FT-save-state**(*s:String*)

```
1: state ← {}
2: for (each feature f in the feature tree) do
3:     add tuple⟨f, f's value⟩ to state
4: end for
5: associate state to identifier s
6: add s to the list of saved states
```

Algorithm 3 Recover system state to a state named *s*

Inputs:
s: state name

Function **FT-recover_state**(*s:String*)

```
1: state ← state associated to s
2: for (each tuple ⟨f, v⟩ in state) do
3:     assign value v to feature f in the feature tree
4: end for
5: remove s from the list of saved states
```

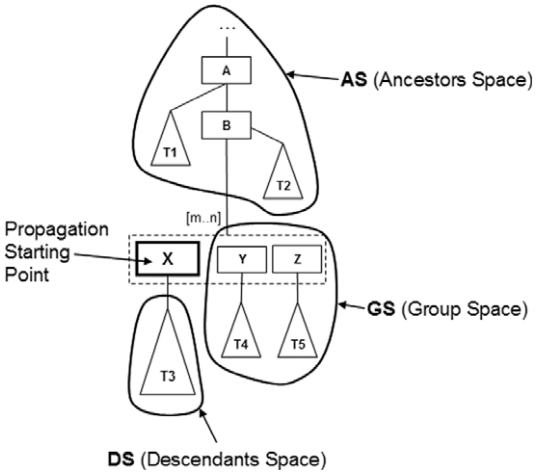

Fig. 7. The three propagation spaces for feature *X*.

Algorithm 4 Propagates a variable assignment throughout the feature tree

Inputs:
f: propagation starting point; *f* is a feature assigned *v*
v: truth value assigned to *f* that starts the propagation

Function **FT-propagate**(*f:feature,v:Boolean*)

```
1: if (f ≠ nil) then
2:     if (v = true) then
3:         propTrueAS(f)
4:         propTrueDS(f)
5:         propTrueGS(f)
6:     else
7:         propFalseAS(f)
8:         propFalseDS(f)
9:         propFalseGS(f)
10:    end if
11: end if
```

Propagating value assignments. Constraint propagation [11] is a very important mechanism used by constraint solvers to enforce consistency and thereby to optimize the search process. For instance, some propagation techniques work by removing values from the domains of the variables that are not part of any solution. This can improve tremendously the efficiency of the solver as unproductive searches are avoided. Propagation is particularly efficient for Boolean domains (unit propagation) since eliminating a value (say *false*) corresponds to assigning the other value to the variable (*true*).

We propose a propagation algorithm tailored to feature trees as shown in Algorithm 4. The algorithm operates on three disjoint "propagation spaces" in the feature tree relative to a particular starting-point feature. Fig. 7 shows the three propagation spaces AS, DS and GS for feature X representing, respectively, the ancestor of X and their children (excluding X), the descendants of X, and the siblings of X within the feature group and their descendants. That is, $AS = \{A, B\} \cup T1 \cup T2$, $DS = T3$, and $GS = \{Y, Z\} \cup T4 \cup T5$. If feature X is not a grouped feature GS is the empty set. The propagation algorithm is recursive and its implementation is simplified by the fact that the propagation spaces examined are disjoint. The *propagate* algorithm is shown in Algorithm 4. The algorithm checks if value v assigned to feature f is *true* or *false* and calls other support operations to propagate the assignment throughout spaces *AS*, *DS*, and *GS*. For a complete implementation of the support algorithms please refer to the Appendix (algorithms *propTrueAS*, *propTrueDS*, *propTrueGS*, *propFalseAS*, *propFalseDS*, and *propFalseGS*).

Algorithm 5 Find the minimum configuration for the feature tree rooted by f

Inputs:
f: root of the feature model
conf: solution containing only *true* features

Function **FT-min-conf**(f : *feature*, *conf* : *feature*{})

```
1:  if (f is the root feature) then
2:     add f to conf
3:  end if
4:  for (each child c of f ) do
5:     if (c is optional) then
6:        skip it...
7:     else if (c is mandatory) then
8:        add c to conf
9:        FT-min-conf(c, conf )
10:    else if (c is a feature group) then
11:       g = 1st child of c
12:       add g to conf
13:       FT-min-conf(g, conf )
14:    end if
15: end for
```

Checking satisfiability. Recall that for a general purpose constraint solver satisfiability is an NP-complete problem. Yet, for feature trees satisfiability can be checked in constant time $O(1)$. In other words, feature trees following the meta-model in Fig. 6 are *always* satisfiable. As a proof, we provide a recursive algorithm called *FT-min-conf* (see Algorithm 5) that always finds a minimum valid configuration for a feature tree. The root of the feature tree f and a variable named *conf* are passed as parameters to the algorithm. Variable *conf* is initially an empty set that will store the features added to the configuration. If f is the root feature, it is added to *conf*. Next, each child feature c of f is visited in pre-order (line 4). Optional child features are skipped and not added to *conf*. Mandatory child features are automatically added to *conf* (lines 7–8) and a recursive call is made to examine each of their children (line 9). If c is a feature group the first grouped feature g is added to *conf* and the others are skipped (lines 10–12). A recursive call is made to examine each of g's children (line 13). All skipped features are assumed *false* and thus not added to *conf*.

Considering that feature trees are always satisfiable the satisfiability operation *FT-is-satisfiable* (algorithm not shown) is trivially implemented by returning the Boolean constant *true*.

Counting solutions. In certain cases counting the number of solutions in a constraint problem can be useful. For instance, in product configuration the number of solutions in the Boolean formula derived from the feature model represents the number of legal configurations in the model. Knowing the number of legal configurations before and after a set of configuration decisions have been made gives an idea of how much the combinatorial space that corresponds to the products in the product line has been shrunk. One can use this information to either continue the manual configuration or to trigger an automated system to automatically complete the current configuration.

Unfortunately, counting solutions using a general-purpose constraint solver is a very time-consuming operation. In fact, in most practical cases computing this operation is infeasible as it requires the solver to visit each and every solution in the problem and yet it is common for configuration problems to have an exponential number of solutions.

Algorithm 6 Count the number of solutions of the (possibly partially instantiated) feature tree rooted by f

Inputs:
f: root of the feature tree
Output:
number of valid configurations in the feature tree rooted by f

Function **FT-count-sol**(*f:feature*) : integer

```
 1: if (f = false ) then
 2:    return 1
 3: end if
 4: count_conf = 1
 5: if ( f is an Exclusive-OR Feature Group) then
 6:    count_conf = 0
 7:    for (each child c of f ) do
 8:       count_conf = count_conf + FT-count-sol(c ) - 1
 9:    end for
10: else
11:    if (f has children) then
12:       for (each child c of f ) do
13:          count_conf = count_conf × FT-count-sol(c)
14:       end for
15:       if (f is Optional or Grouped and f ≠ true) then
16:          count_conf = count_conf + 1
17:       else if (f is an Inclusive-OR Feature Group) then
18:          count_conf = count_conf - 1
19:       end if
20:    else
21:       if (f is Optional or Grouped and f ≠ true) then
22:          count_conf = count_conf + 1
23:       end if
24:    end if
25: end if
26: return count_conf
```

However, because of the special hierarchical arrangement of the variables in a feature tree, it is possible to devise a domain-specific procedure to efficiently count the number of solutions in the feature tree without having to visit each individual solution at a time. This is quite an important property of the feature modelling domain that shows the advantage of a domain-specific procedure over a general one.

The counting procedure takes advantage of the hierarchical arrangement of the variables and relations in the feature tree. For instance, consider a parent feature p, and its optional child features c_1 and c_2. For simplicity, let us assume that neither c_1 nor c_2 have children. We can calculate the number of solutions of the subtree rooted by p by simply multiplying the number of solutions of subtrees c_1 and c_2 which, in this particular case, is four. Notice that it is not necessary to find and combine the solutions of c_1 and c_2 but rather a simple mathematical operation is applied. In addition, since the feature tree is a recursive structure the same strategy can be applied repeatedly. As a consequence, counting configurations in a feature tree can be extremely fast even for feature models containing tens of thousands of features.

We developed an algorithm called *FT-count-sol* (see Algorithm 6) as part of the *FTCS* system to count the number of available solutions in uninstantiated or partially-instantiated feature trees. The algorithm is recursive and based on the ideas previously discussed. The algorithm starts by visiting a given feature f passed as an input parameter and continues by traversing the feature tree in depth-first order. Feature f usually corresponds to the root of the feature tree when the algorithm is first called (non-recursive call). If feature f is *false* (line 1) then there is only one possible configuration for f in which all its descendants are *false* (line 2). Otherwise, if f is not *false* and f is an exclusive-OR feature group (line 5), each of f's children have their configurations counted recursively and the results obtained are added (lines 7–9). The reason for adding the configurations is because only one grouped feature can be *true* in the group while all others are *false* as per the group cardinality [1]. If f is not an exclusive-OR feature group but has children (line 11), then f's childrens' configurations are recursively computed and multiplied (lines 12–14). Multiplication is applied in this context since we need to account for all possible combinations of valid configurations represented in each of f's subtrees. In addition, if f is an optional or a grouped feature and is uninstantiated (in this case we just check condition $f \neq true$ since we have already checked that $f \neq false$) (line 15), we add one more configuration to variable *count_conf* that stores the number of valid configurations for feature f (lines 15–16). This accounts for the case where f and its descendants are all *false*. Instead, if f is an inclusive-OR feature group we deduct one configuration for the opposite reason, i.e., there is no such case where all grouped features are

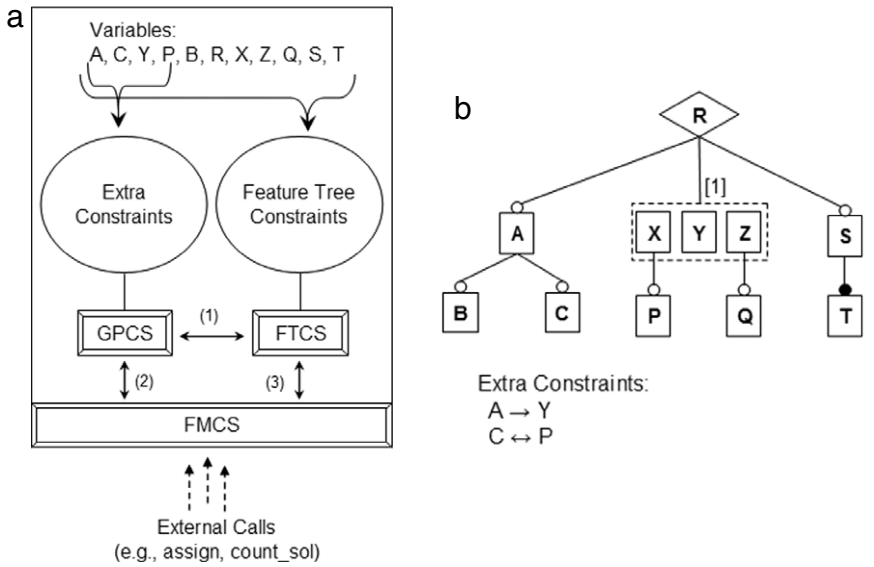

Fig. 8. (a) The architecture of the FMCS and (b) the feature model that illustrates the architecture.

false when their parent feature is *true* (line 17–19). Notice that in all the cases where f is *true* the algorithm computes the valid configurations for f by recursively combining the valid configurations of each of f's *independent* subtrees. In the case of feature groups, group cardinalities have to be considered as well. Finally, if f does not have children but is an optional or grouped feature and uninstantiated (line 21), we add one to variable *count_conf* to account for the case where f is *false* (lines 21–23). This will cause *count_conf* to evaluate to two to represent the two possible assignments to f, i.e., *false* and *true*. For all other cases, f can only be *true* and thus *count_conf* remains one. Line 24 returns the total number of valid configurations for f.

In the worst-case, when none of the features in the feature tree are instantiated, the algorithm *FT-count-sol* visits each feature at most once in depth-first search. As a result, the worst-case complexity of the algorithm is linear in the number of nodes n in the feature tree ($O(n)$). Recall that a general constraint solver would have to perform an exponential number of steps for the same operation ($O(2^n)$) as the solver would typically have to visit every solution.

4.1.2. Integrating the FTCS with a general-purpose constraint solver

As mentioned earlier, the *FTCS* operates exclusively on feature trees and hence cannot be used to reason about feature models that might also contain other arbitrary constraints. A possible solution could be to use a general-purpose solver to address the entire feature model but this would completely ignore the relevant properties of feature trees discussed previously (e.g., guaranteed satisfiability, linear computation of number of solutions).

A much better alternative would be to combine the strength of each solver into a "hybrid" solution. That is, we use a general-purpose constraint solver to address the formula derived from the extra constraints and the *FTCS* to address the feature tree formula and create a layer that integrates the two systems seamlessly. We realize this idea by proposing a hybrid reasoning system for feature models called *FMCS* (Feature Model Constraint System). From this point on, we refer to the general-purpose constraint solver simply as the *GPCS*.

The architecture of the *FMCS* is shown in Fig. 8(a). *GPCS* and *FTCS* are two internal solvers that have their own set of distinct constraints (extra constraints and feature tree constraints, respectively) but share some variables. Since the *FTCS* operates on the feature tree its constraints contain references to all the variables that represent the features of the feature model in Fig. 8(b) . Instead, *GPCS* constraints only refer to 4 of these variables, i.e., A, C, Y and P. As the internal solvers share variables, consistency must be enforced to ensure that constraints of both system are never violated. Numbers (2) and (3) in the figure indicate interactions between the *FMCS* and the internal solvers. There is a direct communication channel between the internal solvers shown in (1). This channel, for instance, allows events generated by the *GPCS* during the search to be handled by the *FTCS*.

In the following, we discuss the dynamics of the *FMCS* and how the system capitalizes on the strengths of both internal solvers to implement the reasoning operations depicted in Table 1 (except for saving and recovering states).

Direct communication between the GPCS and the FTCS. Many modern general-purpose constraint solvers such as [14] use event-based interfaces to communicate internal events. Events allow for an easy extension of the solver's behavior without requiring a deep understanding of its architecture. For instance, by handling events produced by the *GPCS* during the search it is possible for the *FTCS* to modify the dynamics of the search algorithm in *GPCS* by propagating values and raising assignment conflicts. In this context, the *FTCS* is viewed by the *GPCS* as *yet another constraint* that needs to validate

Algorithm 7 Event handler for GPCS variable instantiation events. The GPCS calls this function whenever a variable is added to the search and a value is to be assigned to it

Inputs:
v: instantiated variable

Function **GP-on-instantiating**(*v:variable*)
1: FT-save-state(v)

Algorithm 8 Event handler for GPCS branching events. The GPCS calls this function whenever a new value is tried for a variable during the search

Inputs:
v: branch variable
b: Boolean value assigned

Function **GP-on-branching**(*v:variable, b:Boolean*)
1: FT-restore_state(v)
2: FT-assign(v,b)
3: $P \leftarrow$ FT-propagate(v,b)
4: **if** (an assignment conflict is raised during FTCS propagation) **then**
5: FT-restore-state(v)
6: raises an *assignment conflict exception* to notify the GPCS
7: **else**
8: **for** (each tuple ⟨*variable,value*⟩ in P) **do**
9: GP-instantiate(*variable*,*value*)
10: **end for**
11: **end if**

variable assignments, request propagations, and raise conflicts whenever a constraint in the feature tree is violated. The great advantage of this implementation strategy is that the entire search infrastructure of the *GPCS* is reused without any modification (non-invasive extension).

Algorithms *GP-on-instantiating* (Algorithm 7) and *GP-on-branching* (Algorithm 8) illustrate how this integration can be implemented in practice. The algorithms are developed as event handlers that capture events from the *GPCS*. *GP-on-instantiating* is called whenever a new variable is added to the search tree in the *GPCS*. For each new value tried for the variable (here referred to as *branching*) a call to *GP-on-branching* is made, since a new branch in the search tree is created by the *GPCS*. The implementation of the *GP-on-instantiating* algorithm is straightforward. Essentially, it saves the state of the feature tree giving it the name of the instantiating variable, i.e., prior to any assignments made to the variable (line 1 of (Algorithm 7)). This is necessary in order to allow the feature tree to be restored to a consistent state when the *GPCS* needs to backtrack and try new values for variables.

The algorithm *GP-on-branching* starts by restoring the state of the *FTCS* in order to enforce the consistency between the feature (*FTCS*) tree and the search tree (*GPCS*) (lines 1–2). Next, the branching events from the *GPCS* inform the *FTCS* about value instantiations in the *GPCS* that need to be updated in the feature tree. The *FTCS* performs the updates and propagates the values in the feature tree (line 3–7). If the propagation in the feature tree fails, typically caused by an assignment conflict, an exception is raised to notify the *GPCS* that a constraint in the *FTCS* has been violated (lines 8–10). This will cause the *GPCS* to backtrack and try new values in the search. Otherwise, if the propagation in the *FTCS* succeeds the propagates variables and values are informed to the *GPCS* for proper update (lines 13–15).

Variable assignment and propagation. Assignments and propagations in the *FMCS* are implemented simply as forward calls to the *GPCS* and *FTCS* operations. In addition, special care is needed to enforce the consistency between these two solvers in order to ensure that none of the solver's constraints are ever violated.

Algorithm 9 implements the *FM-assign* operation for the *FMCS* in which value b is assigned to variable v. The algorithm simply forwards the call to the internal solver's operations *FT-assign* and *GP-assign* (lines 1–2). *FM-propagate* propagates variable assignments by updating both internal solvers continuously until either a conflict error is raised or there are no more propagations to carry out. *FM-propagate* is shown in Algorithm 10. The algorithm starts by saving the state of the two internal solvers (lines 1–2) prior to any propagations. This allows the solvers to be restored to a consistent state in case of errors. Next, three sets *FT_tuples*, *GP_tuples* and *tmp_tuples* are defined to support the propagations (lines 3–5). The propagation tuples in the *FTCS* (*GPCS*) are recorded in *FT_tuples* (*GP_tuples*). The *GT_tuples* set is traversed and each of its tuples that represent propagations occurring in the *GPCS* are used to assign values to variables in the *FTCS* (lines 9–17). Similarly, propagations in the *FTCS* stored in the *FT_tuples* set are forwarded to the *GPCS* (lines 18–25). Notice that the *FT_tuples* tuples containing references to variables not found in the *GPCS* are removed from the set (line 11). The propagation loop (line 7) continues

until either a conflict arises in one of the internal solvers (lines 12 and 20) or a consistent state is reached (lines 26–28). The *FMCS* is consistent when the propagation sets *FT_tuples* and *GP_tuples* are empty, i.e., there are no more propagations to carry out. If a conflict error is raised the solvers are restored to their original states (lines 13–14 and 21–22).

Algorithm 9 Assign variable v the truth value b in the FMCS

Inputs:
v: variable
b: Boolean value

Function **FM-assign**(*v:variable, b:Boolean*)
1: FT-assign(v,b)
2: GP-assign(v,b)

Algorithm 10 Propagate value b to variable v in the FMCS

Inputs:
v: variable
b: Boolean value

Function **FM-propagate**(*v:variable, b:Boolean*)
1: FT-save-state("ft-state-before-assignment")
2: GP-save-state("gp-state-before-assignment")
3: *FT_tuples* ← {⟨v, b⟩}
4: *GP_tuples* ← {⟨v, b⟩}
5: *tmp_tuples* ← *nil*
6: *solvers_consistent* ← *false*
7: **while** (*solvers_consistent* is *false*) **do**
8: *tmp_tuples*←*FT_tuples*
9: **for** (each tuple ⟨v, b⟩ in *GP_tuples*) **do**
10: *FT_tuples* ← *FT-propagate*(v, b)
11: Eliminate tuples in *FT* that refer to variables not present in the GPCS
12: **if** (assignment conflicts is FTCS) **then**
13: FT-restore-state("ft-state-before-assignment")
14: GP-restore-state("gp-state-before-assignment")
15: {Error: Assignment Conflict!}
16: **end if**
17: **end for**
18: **for** (each tuple ⟨v, b⟩ in *tmp_tuples*) **do**
19: *GP_tuples* ← *GP-propagate*(v, b)
20: **if** (assignment conflicts in GPCS) **then**
21: FT-restore-state("ft-state-before-assignment")
22: GP-restore-state("gp-state-before-assignment")
23: {Error: Assignment Conflict!}
24: **end if**
25: **end for**
26: **if** (*FT_tuples* = {} and *GP_tuples*= {}) **then**
27: *solvers_consistent* ←*true*
28: **end if**
29: **end while**

Satisfiability and counting solutions. The benefits of using a hybrid solver such as the *FMCS* to reason about feature models are more evident in operations such as checking satisfiability and counting solutions. This is the case because the *FMCS* is able to take advantage of each internal solver's strengths to perform those operations. For instance, while the *GPCS* is very sensitive to the number of variables in the problem (e.g., an extra variable can potentially double the number of steps performed) it provides a quite powerful algorithm for processing unstructured constraints. Meanwhile, the *FTCS* is able to explore properties of the feature modelling domain to provide efficient satisfiability and solution counting operations, but these operations are restricted to constraints expressed as feature trees.

Operation *FM-is-satisfiable* (code not shown) provided by the *FMCS* simply forwards the call to *GPCS* (call to *GP-is-satisfiable* operation). This starts the search procedure in the *GPCS* that will automatically notify the *FTCS* about any events.

Table 2
Performance results of the FMCS and the CSP solver for **satisfiability tests** on various collections of feature models. Timeouts indicate lack of response within 30 sec. Running times are average results of the successful cases.

Solver	Feature model size (20% ECR)						
	500	1000	2000	3000	4000	5000	10,000
CSP solver (Choco 1.2)							
Timeouts [%]	0	20	10	10	50	80	90
Running times [ms]	14	20	61	239	78	53	197
FMCS (Choco as GPCS)							
Timeouts [%]	0	0	0	0	0	0	0
Running times [ms]	15	43	146	248	518	1242	2967

Once a solution in the *GPCS* successfully propagates in *FTCS* the algorithm stops. In theory, the algorithm visits just a subset of the problem variables, as many variables remain uninstantiated in the *FTCS* (feature tree).

Algorithm 11 Count the number of solutions (configurations) of the (possibly partially assigned) feature model rooted by *f*

Inputs:
f: root of the feature model tree

Function **FM-count-sol**(*f:feature*)
1: *num_solutions* ← 0
2: **for** (each solution *S* in the GPCS) **do**
3: *num_solutions* ← *num_solutions* + FT-count-sol(*f*)
4: **end for**
5: **return** *num_solutions*

The rationale of the *FMCS* for counting solutions is to use the *GPCS* to enumerate solutions in the extra constraints and, for each solution found, to prune the feature tree by propagating the variables instantiated by the *GPCS* (see operation *FM-count-sol* in Algorithm 11). Once the feature tree is pruned, the *FMCS* relies on the algorithms provided by the *FTCS* to process reasoning operations. In other words, for each solution found by the *GPCS* and propagated successfully in the feature tree (line 2), the *FTCS* is used to count and accumulate the solutions in the feature tree (line 3). If this operation is repeated for each solution in the *GPCS*, we end up counting the total number of configurations in the feature model (line 5).

4.1.3. Empirical evaluation

We conducted various experiments to evaluate the performance of the hybrid solver *FMCS*. In particular, we aimed at (i) evaluating the gains in performance of the *FMCS* when compared to a general-purpose constraint solver, and (ii) knowing the limits of the *FMCS* in handling very large feature models.

The experiments were run on an AMD Turion system with a 1.6 GHz processor and 1 GB of memory RAM. The constraint programming system Choco 1.2 [14] played two roles: it represented the general-purpose constraint solver in the experiments, and implemented the *GPCS* internal component of the *FMCS* (see Fig. 8(a)). Choco offers a rich set of event-based interfaces, which makes the system a great fit to the *FMCS* architecture described in Section 4.1.2.

The performance and scalability of the *FMCS* and the *CSP solver* were evaluated by means of two operations: *satisfiability* (referred to as *SAT*) and *solution counting* (referred to as *Count-SAT*). The solvers were given 30 sec to complete each operation, otherwise the particular test case was considered unsuccessful. Only consistent models (satisfiable problems) were used in the experiments.

Experiment #1: Satisfiability (SAT). The purpose of this experiment was to measure the performance and the scalability of the *FMCS* and the *CSP solver* in performing *satisfiability checks*.

The experiments were run using 7 collections of 10 satisfiable feature models. The size of the models (number of features) varied from one collection to another, i.e., 500, 1000, 2000, 3000, 4000, 5000, and 10,000 features (20% ECR). In all models the odds for mandatory, optional, inclusive-OR and exclusive-OR features were set to 25%, 35%, 20% and 20%, respectively.

Table 2 shows the performance results for this experiment. The *FMCS* succeeded in 100% of the test cases even for very large models with 10,000 features, with average running time under 3 sec. This was quite surprising since we did not expect much in terms of performance improvements for the *FMCS* for satisfiability checks. In fact, the running times of both solvers were comparable but the number of timeouts of the *CSP solver* were considerably higher. This suggests that the decisions made by the *FMCS* were either quickly propagated or a conflict was found earlier than in the pure CSP solution. This likely avoided many unproductive searches. In addition, the number of decisions made by the *FMCS* was significantly reduced since the solver focused primarily on the extra constraint variables (as opposed to all the variables in the problem), which are usually a fraction of the variables in the feature model. As a consequence, many variables remained uninstantiated even

Table 3
Performance results of the FMCS and the CSP solver for **counting solutions** on various collections of feature models. Timeouts indicate lack of response within 30 sec. Running times are average results of successful cases. All models were satisfiable.

Solver	Feature model size						
	(20% ECR)				(5% ECR)	(2% ECR)	(0% ECR)
	30	50	150	200	500	1000	10,000
CSP solver (Choco 1.2)							
Timeouts [%]	0	80	100	100	100	100	100
Running Times [ms]	2869	9228	–	–	–	–	–
FMCS							
Timeouts [%]	0	0	0	90	40	0	0
Running times [ms]	4.6	9.7	9506	8015	9960	9796	5

after the satisfiability check had been completed. For instance, about 30% of the variables in the models with 5000 features remained uninstantiated upon the completion of the satisfiability checks. Meanwhile, all those variables had to be visited by the *CSP solver*, which certainly affected its performance.

Experiment #2: Counting solutions (Count-SAT). The purpose of this experiment was to measure the performance and the scalability of the *FMCS* and the *CSP solver* in *counting solutions* (legal configurations) in the feature model.

The experiments were run using 7 collections of 10 satisfiable feature models. The size of the models (number of features) varied from one collection to another, i.e., 30, 50, 150 and 200 features with 20% ECR, and other 3 collections with sizes 500 (5% ECR), 1000 (2% ECR), and 10,000 (0% ECR). In all models the odds for mandatory, optional, inclusive-OR and exclusive-OR features were set to 25%, 35%, 20% and 20%, respectively.

Table 3 shows the performance results for this experiment. The *FMCS* succeeded in 100% of the cases (no timeouts) for models with up to 150 features and 20% ECR. Meanwhile, the CSP solver struggled to handle models with 50 features, generating timeouts in 80% of the cases, and failed consistently for larger models. Considering only the cases where no timeouts occurred, the *FMCS* was able to handle models about *5 times larger* than the CSP solver (30 and 150 features, respectively). For models with 200 features and 20% ECR the *FMCS* did not complete counting solutions in 90% of the cases.

The poor performance of the CSP solver in counting solutions was due to the large number of solutions in the feature models analyzed as the solver had to visit each solution individually. Instead, the performance of the *FMCS* is only impacted by the number of solutions in the extra constraints, as the solutions in the feature tree can be counted efficiently by the *FTCS* as discussed in Section 4.1.1.

The experiment also revealed that the *FMCS* can scale well for models with low ECR. Table 3 shows that models with 500 and 1000 features (5% and 2% ECRs, respectively) had their solutions counted by the *FMCS* in 10 seconds or less. In particular, if the counting operation is applied only to the feature tree (i.e., 0% ECR), models with 10,000 features are easily handled (average time of 5ms) (see last table column).

5. Tool support

Fig. 9 depicts CPC, a prototype tool developed to support the plan creation and validation phases of the CPC approach. The Web portal feature model appears loaded in the tool. The feature tree is shown on the left-hand side together with the extra constraints at the bottom. Several configuration spaces have been defined following the scheme shown in Fig. 3. Configuration space *Ad* appears highlighted in the feature tree. The tool allows the splitting of the feature tree into several configuration spaces and the assignment of these configuration spaces to configuration actors. Hypergraph-based techniques are used to identify strong and weak dependencies among configuration spaces and to produce validation rules for CPC plans (see tables Constraints Hypergraph, Conf. Space Hypergraph, and Conf. Space Dependencies). Dependency analysis operations such as $D(n)$, $DT(n)$, and $DF(n)$ support the analysis of feature dependencies on the feature tree and work in conjunction with hypergraph-based techniques (see Feature Basic and Dependency Analysis in Fig. 9). Another prototype tool called ExeCPC is under construction that will allow the development, validation and execution of CPC plans. Plan validation takes into account the validation rules produced by the CPC tool. A critical component of CPC plans is the merging session. A manual merge allows configuration actors to reason about different alternatives to resolve a conflict. Automatic merging algorithms attempt to find a solution to a conflict based on specific strategies. Currently, two strategies are possible. A minimization of changes strategy attempts to find the solution that least changes previous decisions. A priority-based strategy specifies use priorities to decide which decisions should prevail over the others. Automatic merging algorithms have been implemented using the Choco [14] constraint system.

In order to evaluate and compare different approaches to configuration reasoning we developed a tool called 4WATREASON [18]. The tool offers a suite of reasoning approaches and can be easily extended to incorporate new techniques. A screenshot of the tool is depicted in Fig. 10 showing the Web portal feature model loaded (left-hand side). The feature tree and the extra constraints appear in distinct areas as they represent different constraint spaces. In the right-hand side, the three techniques to configuration reasoning discussed in this paper, i.e., the *GPCS*, *FTCS*, and *FMCS* are illustrated. The tool allows any of these techniques to be used to reason about the feature model currently loaded in the tool. For instance,

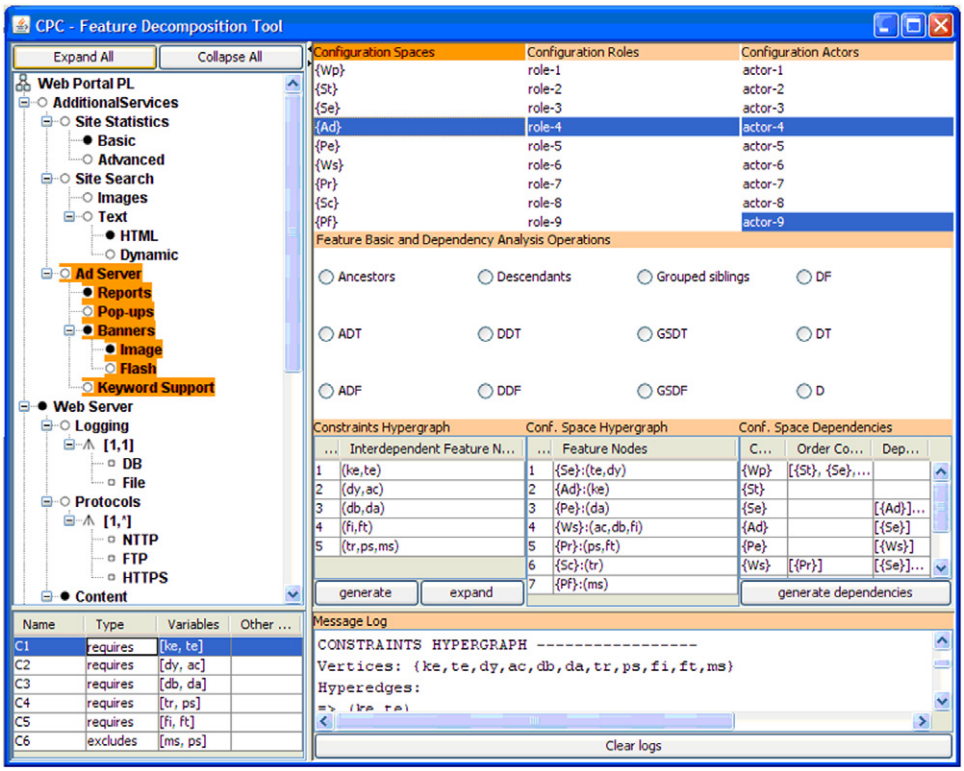

Fig. 9. The CPC tool derives validation rules for CPC plans; The Web portal product line of Fig. 3 is shown loaded in the tool.

the *FMCS* is shown selected in the figure and in the bottom of the window the results for the SAT and Count-SAT operations are indicated. The SAT output "yes" indicates that the feature model is satisfiable. Count-SAT shows that there are 7,004,160 possible valid configurations for the Web portal feature model. In addition, the user is allowed to click on the feature tree to select or deselect features, i.e., manually configure the feature model. Recall that the reasoning operations can still be applied for partially configured feature models. Several other functionalities are available in the tool, such as those to manipulate binary decision diagrams [19,20], but it is beyond the scope of this paper to discuss them. The APIs developed to support the 4WATREASON tool were also used to build the testing tools that supported the experiments reported in Section 4.1.3.

6. Related work

Product configuration has also been studied in artificial intelligence as a constraint satisfaction problem (CSP). In this framework, configuration knowledge is described as a component-port representation [21] that uses constraints to restrict the way components can be combined. Constraints are usually written in formal notation (e.g., first-order logic). Similarly, user requirements are translated to a formal representation allowing the configuration problem to be solved fully by automated systems known as configurators. Alternatively, configuration can be encoded as a distributed problem [21]. In distributed configurations, the problem is translated into a distributed constraint satisfiability problem (DisCSP) [22] in which the constraints and variables are fragmented over multiple configuration environments. Each environment is controlled by an intelligent software agent that works as a local configuration system. DisCSP approaches build on distributed algorithms to support agent communication (e.g., message passing mechanisms) and coordination (e.g., constraint enforcement). CSP and DisCSP focus on developing algorithms and machinery support for solving constraint satisfaction problems. The assumption is that machines can quickly process thousands of instructions and perform efficient backtracking until a desirable solution is found. The involvement of humans in the process is limited to providing requirements to the configuration system in terms of logic formulas. Instead, in our approach the major goal is to support the coordination of human decision-making in product configuration. Tool support is provided, not as a means to solve the problem, but to provide assistance for humans to carry out the job themselves.

Staged configuration [23] was an initial starting point for our work as it explored various scenarios in which product configuration is carried out by multiple configuration actors through different stages. The authors introduced two configuration techniques called specialization and multi-level configuration to support the progressive configuration of products. The CPC approach relates to staged configuration in at least two points. First, it furthers the discussion regarding decision-making coordination in collaborative configuration, i.e., how the work of multiple people working on the same feature model can be organized, and conflicts avoided or minimized. Second, it addresses the issue of automated support by

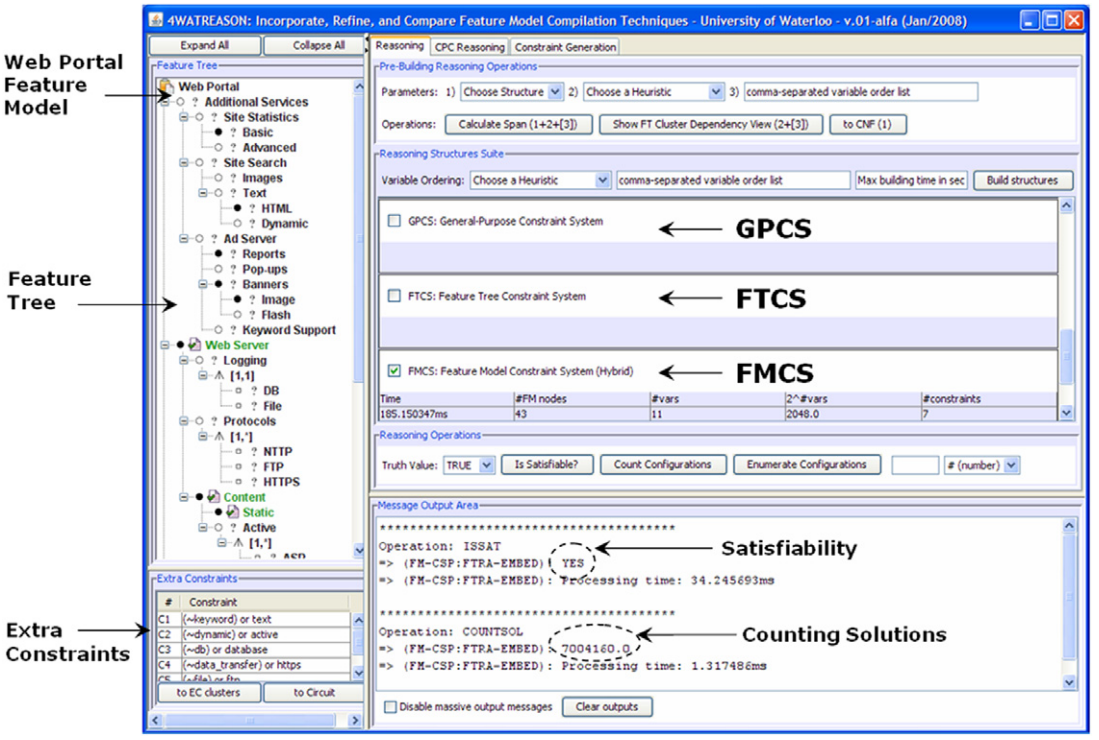

Fig. 10. 4WATREASON - A suite of reasoning approaches for feature models and product configuration.

exploring possible techniques to reason about feature models and product configuration, including the proposal of efficient algorithms tailored to the feature modelling domain.

Several techniques for the automated analysis of feature models have been studied in the literature [24–26]. They typically translate feature models to a specific formal encoding such as a Boolean formula or a binary decision diagram and propose the use of general off-the-shelf systems (e.g., constraint solver, binary decision diagram engine) to reason about the feature model. Our research gives a step forward by investigating how existing off-the-shelf systems can be combined with algorithms tailored to the feature modelling domain in order to improve the overall performance of reasoning operations. For instance, we have introduced a domain-specific constraint solver called *FTCS* that takes advantage of particular properties of the feature modelling domain to implement efficient algorithms for feature trees that can be orders of magnitude faster than a standard solution. In addition, we showed that the *FTCS* system can be further integrated with a general constraint solver to form a hybrid reasoning system (called *FMCS*) that covers the entire feature model.

Finally, the work from Czarnecki et al. [27] proposes a procedure to extract feature models from arbitrary Boolean formulas. The extraction algorithms attempt to identify as many relations as possible that can be represented together as a single feature tree structure and additionally attaches unstructured relations as extra constraints to that structure. We see an interesting interplay between the work of Czarnecki and ours. One can use the extraction procedure to build a feature model for an arbitrary formula and afterwards take advantage of the hybrid solver we proposed (*FMCS*) to reason about the extracted model.

7. Conclusion

In this paper we revisited our approach to collaborative product configuration (CPC) and showed how it can be used to coordinate human decision-making in feature-based configuration. We discussed how the approach allows the decisions in a feature model to be split into smaller more-manageable units called configuration spaces and how configuration spaces can be arranged into configuration sessions to form a process model named configuration plan. We illustrated how configuration plans define the order of execution of the configuration sessions (e.g., sequential, concurrent) and allow the specification of merging sessions to manage decision conflicts that eventually arise during configuration. As discussed, configuration plans are expressed as a workflow-like structure and are validated according to a set of dependency rules.

In addition, the paper proposed a set of reasoning algorithms tailored to the feature modelling domain packed in the form of a domain-specific constraint system for feature trees called *FTCS*. We showed how the *FTCS* could be integrated to a general-purpose constraint solver to form a hybrid reasoning system for feature models called *FMCS*. We compared the performance of the *FMCS* with that of a general-purpose solver and showed that the former can be much more efficient than the latter, especially for feature models with low ECR and for specific operations such as counting the number of solutions in the feature model.

The paper furthers the understanding of collaborative configuration and its major challenges as well as the required infrastructure to support the product configuration process. We hope that the algorithms and the insights can encourage the development of many new reasoning algorithms for feature models in the future. In particular, we are interested in building a library of such efficient algorithms to improve automated support for our proposed CPC approach.

Future work includes experimenting with new techniques to support efficient reasoning about product configuration, such as the use of binary decision diagrams [19,20]. Moreover, we intend to explore other reasoning operations such as detecting "dead" features, providing explanations for feature model inconsistency (unsatisfiability), and finding optimal or approximate solutions for decision conflicts.

Appendix. Constraint propagation algorithms for feature trees operating on different propagation spaces

Algorithm 12 Propagates a *true* assignment UP in the feature tree

Inputs:
var: propagation starting point; *var* is a feature assigned *true*

Function **propTrueAS**(*var*)

```
1:  if (var≠ nil) then
2:    parent ← parent(var)
3:    if (parent≠ nil and parent is NOT the root node and parent is unassigned) then
4:      assign(parent,true)
5:      if (var is NOT a grouped feature) then
6:        propTrueDS(parent, var)
7:        if (parent is a grouped feature ) then
8:          propTrueGS(parent)
9:        end if
10:     end if
11:     propTrueAS(parent) {recursive call}
12:   end if
13: end if
```

Algorithm 13 Propagates a *true* assignment DOWN in the feature tree

Inputs:
var: propagation starting point; *var* is a feature assigned *true*
evar: child of *var* that will be excluded from propagation

Function **propTrueDS**(*var*, *evar*)

```
1:  if (var≠ nil) then
2:    if (var is a feature group) then
3:      if (the sum of true and unassigned features in the group equals the group lower bound) then
4:        for (each unassigned feature G) do
5:          if (G ≠ evar) then
6:            assign(G, true)
7:            propTrueDS(G,evar)
8:          end if
9:        end for
10:     end if
11:   else
12:     for (each child feature C of var) do
13:       if (C ≠ evar) then
14:         if (C is a feature group w/ lower bound > 1 or a mandatory feature) then
15:           assign(C, true)
16:           propTrueDS(C,evar)
17:         end if
18:       end if
19:     end for
20:   end if
21: end if
```

Algorithm 14 Propagates a *true* assignment within a feature group

Inputs:
var: propagation starting point; *var* is a grouped feature assigned *true*

Function **propTrueGS**(*var*)

```
1: if (var ≠ nil and var is a grouped feature) then
2:   if (number of grouped features assigned true is equal to group upper bound) then
3:     for (each unassigned grouped feature G in the group) do
4:       assign(G, false)
5:       propFalseDS(G, nil)
6:     end for
7:   else if (the sum of true and unassigned grouped features is equal to the group lower bound) then
8:     for (each unassigned grouped feature G in the group) do
9:       assign(G, true)
10:      propTrueDS(G, nil)
11:    end for
12:  end if
13: end if
```

Algorithm 15 Propagates a *false* assignment UP in the feature tree

Inputs:
var: propagation starting point; *var* is a grouped feature assigned FALSE

Function **propFalseAS**(*var*)

```
1: if (var≠ nil) then
2:   parent ← parent(var)
3:   if (parent≠ nil and parent is NOT the root node) then
4:     if (var is mandatory feature or a feature group with lower bound > 0) then
5:       assign(parent,false)
6:       propFalseDS(parent, var)
7:       if (parent is a grouped feature) then
8:         propFalseGS(parent)
9:       end if
10:      propFalseAS(parent) {recursive call}
11:    end if
12:  else if (var is a grouped feature) then
13:    if (number of grouped features assigned false > (group size - lower bound)) then
14:      assign(parent, false)
15:      propFalseAS(parent)
16:    end if
17:  end if
18: end if
```

Algorithm 16 Propagates a *false* assignment within a feature group

Inputs:
var: propagation starting point; *var* is a grouped feature assigned FALSE

Function **propFalseGS**(*var*)

```
1: if (var≠ nil and var is a grouped feature) then
2:   if (number of true and unassigned features is equal to group lower bound) and (parent(var) is true) then
3:     for (each unassigned grouped feature G in the group) do
4:       assign(G, true)
5:       propFalseDS(G, nil)
6:     end for
7:   else if (number of FALSE-assigned features is greater than group upper bound) then
8:     for (each unassigned grouped feature G in the group) do
9:       assign(G, false)
10:      propTrueDS(G, nil)
11:    end for
12:  end if
13: end if
```

Algorithm 17 Propagates a *false* assignment DOWN in the feature tree

Inputs:
var: propagation starting point; *var* is a feature assigned *false*
evar: child of *var* that will be excluded from propagation

Function **propFalseDS**(*var*, *evar*)

```
1: if (var≠nil) then
2:   for (each child feature C of var) do
3:     if (C ≠ evar and C is unassigned) then
4:       assign(C, false)
5:       propFalseDS(C,evar)
6:     end if
7:   end for
8: end if
```

References

[1] P. Clements, L. Northrop, Software Product Lines: Practices and Patterns, Addison-Wesley, Boston, MA, 2001.
[2] K. Pohl, G. Böckle, F. J. van der Linden, Software Product Line Engineering: Foundations, Principles and Techniques, Springer-Verlag New York, Inc, Secaucus, NJ, USA, 2005.
[3] K. Kang, S. Cohen, J. Hess, W. Nowak, S. Peterson, Feature-oriented domain analysis (FODA) feasibility study, Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, November 1990.
[4] K. Czarnecki, U.W. Eisenecker, Generative Programming: Methods, Tools, and Applications, Addison-Wesley, Boston, MA, 2000.
[5] V. Cechticky, A. Pasetti, O. Rohlik, W. Schaufelberger, XML-based feature modelling, in: J. Bosch, C. Krueger (Eds.), Software Reuse: Methods, Techniques and Tools: 8th Int. Conference, ICSR 2004, Madrid, Spain, July 5–9, 2009. Proceedings, in: Lecture Notes in Computer Science, vol. 3107, Springer-Verlag, Heidelberg, Germany, 2004, pp. 101–114.
[6] Kyo Kang, and Kwanwoo Lee, JaeJoon Lee, FOPLE — Feature Oriented Product Line Software Engineering: Principles and Guidelines, website: http://www.sei.cmu.edu/productlines, 2002.
[7] C.W. Krueger, Software mass customization, white paper (Oct. 2001).
[8] D. Beuche, pure::variants Eclipse Plugin, user Guide. pure-systems GmbH. Available from http://web.pure-systems.com/fileadmin/downloads/pv_userguide.pdf, 2004.
[9] M. Antkiewicz, K. Czarnecki, FeaturePlugin: Feature modeling plug-in for Eclipse, in: OOPSLA'04 Eclipse Technology eXchange (ETX) Workshop, 2004, paper available from http://www.swen.uwaterloo.ca/~kczarnec/etx04.pdf. Software available from http://gp.uwaterloo.ca/fmp.
[10] D. Batory, D. Benavides, A. Ruiz-Cortes, Automated analysis of feature models: Challenges ahead, Commun. ACM 49 (12) (2006) 45–47. doi:10.1145/1183236.1183264.
[11] F. Rossi, P. van Beek, T. Walsh, Handbook of Constraint Programming, Elsevier Science, 2006.
[12] S.A. Cook, The complexity of theorem-proving procedures, in: STOC'71: Proceedings of the third annual ACM symposium on Theory of computing, ACM, New York, NY, USA, 1971, pp. 151–158. doi:10.1145/800157.805047.
[13] M. Mendonca, T. Bartolomei, D. Cowan, Decision-making coordination in collaborative product configuration, in: 23rd Annual ACM Symposium on Applied Computing, ACM, 2008.
[14] Cambazard Hadrien, et al. Choco: constraint programming system, website: http://sourceforge.net/projects/choco/, 2003.
[15] D.L. Berre, A. Parrain, O. Roussel, L. Sais, SAT4J: A satisfiability library for Java http://www.objectweb.org/phorum/download.php/16,291/sat4j-D-Le-Berre.pdf, 2005.
[16] D.S. Batory, Feature models, grammars, and propositional formulas, in: Software Product Lines, 9th Int. Conference, SPLC 2005, Rennes, France, September 26-29, 2005, Proceedings, in: LNCS, vol. 3714, Springer, 2005, pp. 7–20.
[17] M. Mendonca, T. Oliveira, D. Cowan, Collaborative product configuration in software product lines: Formalization and dependency analysis, Journal of Software 3 (2) (2008) 69–82. URL http://www.academypublisher.com/jsw/vol03/no02/jsw03026982.pdf.
[18] M. Mendonca, Research project web-page for 'efficient compilation techniques for large scale feature models', available at: http://csg.uwaterloo.ca/~marcilio/fmcompilation/index.html, 2008.
[19] R.E. Bryant, Graph-based algorithms for boolean function manipulation, IEEE Transactions on Computers 35 (8) (1986) 677–691.
[20] C. Meinel, T. Theobald, Algorithms and Data Structures in VLSI Design, Springer-Verlag, 1998.
[21] A. Felfernig, G. Friedrich, D. Jannach, M. Zanker, Towards distributed configuration, in: KI'01: Proceedings of the Joint German/Austrian Conference on AI, Springer-Verlag, London, UK, 2001, pp. 198–212.
[22] M. Yokoo, E. Durfee, T. Ishida, K. Kuwabara, The distributed constraint satisfaction problem: Formalization and algorithms, Knowledge and Data Engineering IEEE Transactions on 10 (5) (1998) 673–685. doi:10.1109/69.729707.
[23] K. Czarnecki, S. Helsen, U. Eisenecker, Staged configuration through specialization and multi-level configuration of feature models, Software Process Improvement and Practice 10 (2) (2005) 143–169. http://swen.uwaterloo.ca/~kczarnec/spip05b.pdf.
[24] M. Mendonca, A. Wasowski, K. Czarnecki, D.D. Cowan, Efficient compilation techniques for large scale feature models, in: Int. Conference on Generative Programming and Component Engineering, GPCE'08, 2008, pp. 13–22.
[25] D. Batory, Feature models, grammars, and propositional formulas, Tech. Rep. TR-05-14, University of Texas at Austin, Texas, March 2005.
[26] D. Benavides, P. Trinidad, A. Ruiz-Cortes, Automated reasoning on feature models, in: Proceedings of the 17th Conference on Advanced Information Systems Engineering, CAiSE'05, Porto, Portugal, 2005, in: LNCS, Springer, 2005.
[27] K. Czarnecki, A. Wasowski, Feature models and logics: There and back again, in: Proceedings of 10th Int. Software Product Line Conference, SPLC 2007, IEEE Press, 2007.